EXHIBIT 5

Proceedings

IEEE INFOCOM '98

The Conference on Computer Communications

Volume 2

Seventeenth Annual Joint Conference
of the IEEE Computer and Communications Societies

Gateway to the 21st Century

29 March - 2 April 1998
Hotel Nikko, San Francisco, CA, USA

INFOCOM'98 gratefully acknowledges the generous travel support provided by the National Science Foundation.

Sponsored by
IEEE Computer Society
IEEE Communications Society

IEEE

IEEE COMMUNICATIONS SOCIETY

Proceedings IEEE INFOCOM'98 The Conference on Computer Communications

IEEE Catalog Number: 98CH36169 (softbound)
98CB36169 (casebound)

ISBN Softbound: 0-7803-4383-2
Casebound: 0-7803-4384-0
Microfiche: 0-7803-4385-9
CD-Rom: 0-7803-4386-7

ISSN 0743-166X

Additional copies of this publication are available from

IEEE Operations Center
P. O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331 USA

1-800-678-IEEE
1-908-981-1393
1-908-981-9667 (FAX)
833-233 (Telex)
email: customer.services@ieee.org

DAN: Distributed Code Caching for Active Networks

Dan Decasper[1], *Bernhard Plattner* [2]
dan@arl.wustl.edu, plattner@tik.ee.ethz.ch
[1]*Applied Reserach Laboratory, Washington University, St.Louis, MO, USA*
[2]*Computer Engineering and Network Laborator, ETH Zurich, Switzerland*

ABSTRACT

Active networking allows the network infrastructure to be programmable. Recent research focuses on two commonly separated approaches: "capsules" and "programmable switches". Capsules are typically small programs in packets which flow through the network and are executed in-band on nodes receiving them. Programmable switches are network devices which offer a back door to inject code by a network administrator out-of-band in order to enhance the device's capabilities. By combining these two approaches, this paper proposes a novel system architecture which allows both application specific data processing in network nodes as well as rapid deployment of new network protocol implementations. Instead of carrying code, data packets carry pointers to digitally signed active modules initially loaded on-the-fly, in-band, from trusted code servers on the network. Packet processing runs at high speed, may access and modify the whole network subsystem and no potentially slow virtual machines are needed.

Key words*: active networks; application specific data processing; automatic protocol deployment; distributed code caching*

1. INTRODUCTION

Active networks [24] are packet-switched networks in which packets can contain code fragments that are executed on the intermediary nodes. The code carried by the packet may extend and modify the network infrastructure. The goal of active network research is to develop mechanisms to increase the flexibility and customizability of the network and to accelerate the pace at which network software is deployed. Applications running on end systems are allowed to inject code into the network to change the network's behavior to their favor.

Increased flexibility and customizability implies security problems. Stability is crucial to network devices not only because the network has become so important to people's daily work, but also because the devices lie in separate administrative domains and are run by different people. Breakdowns are extremely hard to trace and fix.

In this paper, we present an active network architecture which addresses security through policy and simple cryptology instead of potentially slow, restricting and complicated technical measures like virtual machines. We plan to implement our solution on a platform specially designed for active networking through Distributed Code Caching (DAN). We will show examples for both safe, new network protocol deployment as well as application specific data processing in network nodes using our system.

The next section, section 2, explains the main motivations and problems of active networks and puts the security issue into a different light. Section 3 describes our system and addresses questions and concerns related to it. Section 4 outlines two applications we plan to implement as proof-of-concept for our system. Section 5 provides some insight into the actual implementation environment. Section 6 relates our work to that of others, and Section 7 summarizes our ideas.

WARNING: this paper is a design paper. It elaborates on a idea which has not been implemented nor verified yet. We do not provide final results. Several components of the system described may be subject to chance!

2. ACTIVE NETWORKS: MOTIVATION AND PROBLEMS

Active networking is an exciting area of research which concentrates on two commonly separated approaches: "programmable switches" [2, 6, 22] and "capsules" [18, 25]. These two approaches can be viewed as the two extremes in terms of program code injection into network nodes.

Programmable switches typically "learn" by implicit, out-of-band injection of code by a network administrator. Research in the area of programmable switches either focuses on how to upgrade network devices at run time or on upgrades which support end system applications (e.g. congestion control for real-time data streams) or on a combination of both.

Capsules, though, are miniature programs that are transmitted in-band and executed at each node along the capsule's path. This approach introduces a totally new paradigm to packet switched networks: instead of "passively" forwarding data packets, routers execute the packet's code and the result of that computation determines what happens next to the packet. It looks like this approach has an enormous potential impact for the future of networking. In the near future, capsule-based solutions potentially suffer from performance related problems mainly due to security constrains. They commonly make use of a virtual machine that interprets the capsule's code to safely execute on a node. This is similar to the way Java applets run in web browsers. The virtual machines must restrict the address space a particular capsules might access to ensure security, which restricts the application of capsules.

One of the main motivations of active networks is to reduce the difficulty of integrating new technologies and standards into a shared network infrastructure. If one follows the development and deployment efforts in the context of the IPv6 [11] network protocol (successor of IPv4), one realizes how extremely difficult such a project is in today's heavily used Internet. The design efforts for IPv6 started as early as 1994 [7] and still even the most optimistic voices do not expect IPv6 to be used widely before 2005-2010. Since it is so extremely hard (if not impossible) to change IPv6 globally with the common network architecture once it starts getting used, every single feature must be discussed very carefully and in a lot of detail. The discussion on how to use the IPv6 class (former: 'priority') field has been ongoing for months and still no full consensus is reached. Besides technical issues, political arguments more and more influence the design and deployment of such technologies since they became strategic for most companies. What is needed is a fully automated way to deploy and revise new network protocols globally. This allows for incremental refinement of specifications and implementations based on real-world experience which has not been possible so far. The system proposed in this paper shows a possible architecture to achieve that.

Even if protocols like IPv6 and TCP/UDP over IPv6 are spread and used worldwide, they will be the "best" network protocols for a subset of applications only. The emerging need for fast multimedia applications shows that it is often very desirable to run application specific data processing in network nodes. [5] shows that active networks technologies successfully help to handle congestion which occurs when transmitting MPEG video streams over loaded routers by application specific, selective packet dropping. Other examples show how to use the technology for reliable multicast [16, 17], mixing of sensor data [15], web-cashing [7], and many more. This is another important application of active networks technologies, which we focus on.

To overcome the performance related problems of capsules, we think that a combination of both the capsule and the programmable switch approach is very appealing. We address security through policy and cryptology. If the node executing an active code fragment has the possibility to reliably check the source of a fragment and the identity of it's developer, it is able to decide based on that information whether or not it wants to accept and execute the fragment. This is very similar to the way software is distributed to end systems. Nobody would want to run a word processor in a virtual machine just because he/she can't trust the word processor's developer. By knowing the origin of an application, we implicitly assume that it does not corrupt our system. The code fragments used in our system are digitally signed by the developer and come from trusted code servers. This introduces some restrictions regarding the authorship and the source of active network code for the benefit of security and performance, but we believe this to be an appropriate compromise.

3. THE DAN ARCHITECTURE

In common networks, network nodes are connected over (possibly) heterogeneous link layer media to each other. They are able to talk together because they use a common network layer protocol (typically IP) which hides both the data link layer and the transport and higher layer details. Every node which is forwarding a packet is called a "router" and does not have to know about protocols on layers higher than the network layer. A packet usually flows from one end system called a "host" to one or multiple other hosts by jumping from router to router until it reaches the final destination. Most of the communication in the internet is client-server oriented: one node called the "client" sends a request for some sort of "passive"[1] data to another node called "server". The server may or may not, depending on policies established on the server, provide the client with the data. In this paper we call such a server "data server" in order to distinguish it from "code servers" introduced later.

Network packets consist of a finite sequence of identifiers for functions and input parameters for these functions. The functions are normally daisy-chained in the sense that one function calls the next according to the order of the identifiers in the data packet. The first function is determined by the hardware (the interface the packet is received on) and the last function or set of functions is implemented in the application consuming the packet. An Ethernet packet, for example, contains a unique identifier for the upper laying protocol (0x0800 for IPv4, 0x08dd for IPv6). By demultiplexing an incoming packet on this value, the kernel decides to which function or set of functions the packet gets passed next. Each of the functions may also decide not to call the next function for several reasons (e.g. forwarding the packet to the next hop and thereby skipping over transport layer data or detection of errors). Depending on the type of node the packet is processed on and the packet's content, only a subset of the functions may be called.

This scheme is common to all of the well known types of networks. We introduce a minimal amount of formalism to describe this scheme inspired by [6] which uses similar terminology. A packet can be interpreted as a sequence of function identifier $fi_2 \ldots fi_N$ with usually distinct sets of parameters $P_1 \ldots P_N$ (Figure 1). The function identifiers and parameters are strictly order in the

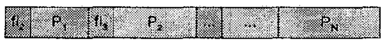

Figure 1: Datagram (schematically)

packet with fi_n followed by P_{n-1} with the exception of the final parameter P_N ($fi_2P_1 \ldots P_N$, ignoring the for the discussion irrelevant detail that the function identifier must not necessarily always be the first byte or word of the fi_nP_{n-1} pair). The first function is not indicated by any function identifier but derived from the context in which packet processing starts (e.g. the packet is received by an Ethernet card, therefore the Ethernet input function is called). Every function identifier fi_n unambiguously points to an implementation of the function f_x ($f_x \mid x=fi_n$). If the function identified by fi_{n-1} decides to call the function identified by fi_n and fi_n[2] is not known to fi_{n-1}, fi_{n-1} proceeds into some form of error handling by usually dropping the packet and possibly notifying the node originating the packet.

[1]"Passive" meaning here that the content of the data is irrelevant to the network subsystem of the client, the data server and the intermediate routers

[2]we say "fi_{n-1}" and mean "the implementation of the function identified by fi_{n-1}"

Now, we simply add a new alternative for the node's behavior. If the node is unable to locate the function identified by fi_n, it temporary suspends processing of the packet and calls a "code server" for the implementation of the function f_x. In contrast to data servers which provide a client with "passive" data, a code server is a well known node in the network which provides a library of (possibly) unrelated functions for (possibly) different types of operating systems from various developers. We call a router or end-system modified to download active modules an Active Network Node (ANN). Figure 2 shows an example of a client downloading

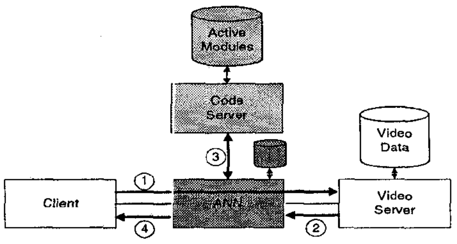

Figure 2: ANN fetching active module from code server

real-time video through an ANN which involves several steps: (1) The ANN receives the connection setup request and forwards it to the video sever; (2) the video server replies with a packet referencing a function for real-time video transport; (3) the ANN does not have the code referenced in it's local cache and therefore contacts a code server for the module; (4) the ANN receives the active module, dynamically links it in its networking subsystem, applies the data to the transport function and forwards the packet to the client. Once the module is downloaded, it is permanently stored locally on the ANN preventing other downloads for the same active module in the future. One can think of the code server as a "cache" for the active module code. By putting code servers into a hierarchy (see section 3.1.2), several levels of caching can be established.

The cycle can be described in pseudo-code as follows:

```
execute the first function depending on hardware properties (interface the packet is
received on)
n=2
do {
    x=$fi_n$
    if($f_x$ not locally available) {
            enqueue packet
            send request for active module to code server
            resume packet processing with a packet in input queue
    } else {
l1:         execute $f_x(P_n)$
    }
    inc n
}
on reception of active module implementing $f_x$ {
    dequeue packet
    goto l1
}
```

Eventually one of the functions f_n will jump out of the 'do' loop and terminate processing of the packet either by dropping the packet, forwarding the packet to another node or passing it to an application.

It is important to note that the active modules offered by the code server are programmed in a higher level language such as 'C' and compiled into object coded for the ANN platform. Once the functions are loaded by the node, they are in no way different than the ones compiled into the network subsystem at build-time. For example, the functions have as much control over the network subsystem's data structures as any other function in the same context, they are executed as fast as any other code.

Another important point in this context is security: by actually loading active modules from well known code servers which authenticate them and give the node the possibility to check the module's sources, and by providing digitally signed modules from well known developers only, the whole security problem advanced active networks solutions usually suffer from, degenerates to the installation of a simple rule on the node which let it choose the right code server (see section 3.1.1 for explanation on how exactly the node chooses the code server) and a database of public keys to check the developers signature. The node uses RSA public key encryption [21] to check signatures and authentication information. It automatically downloads the public keys required through the Domain Name System (DNS) featuring its recently published security extensions [13]. Only one public key (usually the networking subsystem developer's key) has to be installed initially on the node. Alternatively, if a node relies only on code server authentication, it can use IP security [5] which implementation is mandatory in IPv6 and does not require any additional overhead for security.

The combined facts that the active modules may have access to the whole network subsystem and that they run at high speed, we believe that our solution offers a powerful environment for active networks. While our approach is safe and fast, it may appear less flexible than the typical capsule approach. This is true to a certain degree. In order to properly authenticate code modules, some administrative overhead must be introduced. An independent authority must take care of distributing function identifiers (as it is already the case today), developer codes, and network subsystem code in similar ways IP addresses or MAC addresses are distributed to developers and sites. We believe that active modules will be implemented mainly by companies developing network subsystems which are not very large in number. Thus, we don't expect this kind of limitation to be very important in a real-world scenario.

3.1 DAN properties

There are (other) problems introduced with on-demand, just-in-time fetching of active modules described above which have to be addressed. This section discusses the most important issues. We elaborate on where the active modules come from and how a node may find a code server. We suggest putting code servers in a hierarchy for best possible distribution of active modules in the large scale. Policies installed locally on nodes regulate storage and acceptance issues. Finally, interaction of the most common network protocols (TCP/UDP/IP) with our system and the problem of connection setup delay due to code downloading is discussed.